# SPANNER:
# GOOGLE'S SCALABLE AND FAULT TOLERANT TRANSACTIONAL DATABASE

Over the past few lectures we learned about key mechanisms and protocols for achieving scalability and fault tolerance in a strong-semantic, transactional database. Our focus on strong-semantic, transactional databases was motivated by the ease of programming with a clean abstraction such as ACID transactions: groups of operations that are executed atomically and in isolation from other ongoing groups. We studied key mechanisms for achieving ACID (mainly atomicity and isolation) in the non-distributed case: write-ahead logging (WAL) and locking, respectively. Then we talked about how to achieve these same properties in a distributed setting: where we want to: (1) shard the database across multiple servers so we can increase the overall system's capacity; and (2) replicate the shards across multiple servers so we can tolerate failures. Specifically, we talked about (1) two-phase commit (2PC), a protocol that is needed to achieve atomicity in a sharded database; and (2) RAFT, a protocol that is needed to keep the WAL consistent among the replicas of each shard.

Today we will look at the design and implementation of Spanner, Google's scalable and fault-tolerant transactional database. It leverages and combines all of the mechanisms above (or variants thereof), and adds an additional component: a notion of real time, which is used to offer a semantic even stronger than what's typically implied by distributed ACID transactional systems. We'll first look at Spanner's architecture and how it combines the mechanisms we've already studied, then we'll overview the real time component and show how Spanner uses it. We'll then be ready to watch the Spanner technical talk from OSDI ☺.

## I. Spanner Architecture

Figure 1 shows Spanner's high-level architecture. A Spanner database stores one or more relational tables. Tables must have a primary key. Each table is sharded into *tablets* (Spanner's name for *shards*) along the primary key. Each tablet therefore corresponds to a sequence of rows in a table and has a start_row and an end_row associated with it. The tablet's state – consisting of files storing the tablet's checkpoints on disk plus its WAL – are stored in a distributed file system, called Colossus. We didn't look into the design of Colossus in this class, but you shouldn't have to worry about it much here: all of the transactional properties are achieved through Spanner and not part of Colossus.

Each tablet is managed and served by three tablet servers, which replicate its state across multiple, geographically distributed databases. The set of tablet servers replicating one tablet is called its "replica group." The replica group may differ for each tablet. Figure 1 shows the structure of one replica group. The tablet servers within a replica group coordinate using Paxos, a

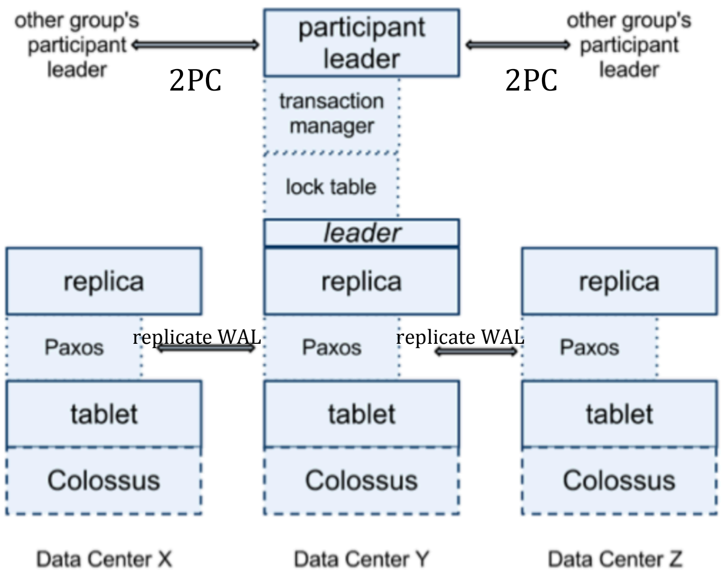


Figure 1. Spanner Architecture.

consensus algorithm akin to **RAFT**, to replicate the WALs they maintain for their tablet. As in RAFT, one of the tablet servers in each group is promoted to the rank of *leader* for some period of time. The other replicas are "followers." The leader tablet server will be responsible for taking in clients' operations for the tablet (e.g., reads from and updates to the tablet as part of transactions), grabbing the appropriate **locks** to ensure isolation from concurrent transactions, and logging entries to the replicated **WAL** to get transactional semantics and fault tolerance at the level of individual tablets.

Across tablets, the leaders of the replica groups involved in a transaction coordinate using **2PC** to preserve atomicity despite sharding. For that, the leader of one replica group will become the coordinator, and the others will be the participants in 2PC. Recall that the 2PC mechanism on its own behaves very poorly if one of the participants, and especially the coordinator, becomes unresponsive or disconnected from the others at an inopportune moment. A large part of the system can grind to a halt if that happens. In Spanner, because 2PC is used in conjunction with Paxos, this situation is avoided. Specifically, the leader of each shard is actually backed by an entire group of replicas distributed across multiple geographies, which are kept up to date w.r.t. the tablet's state using Paxos, and hence any of these replicas can take as leaders of their group should something happen to the current leader.

With these mechanisms (plus a few more to permit for example the expansion and repartition of tablets), Spanner provides some pretty strong data management semantics: atomic transactions that can be performed at scale and with geographical fault tolerance.

What about performance? 2PC, especially across geographically replicated groups, is expensive! To cite the Spanner authors: "[S]ome authors have claimed that two-phase commit is too expensive to support, because of the performance or availability problems that it brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions." So yes, it's expensive, but from a usability perspective, it's worth it: please keep that in mind when you decide on your own designs or the choice of backend system for your own applications.

## II. Even Stronger Semantics

It turns out that ACID transactional semantics were insufficient for Google's semantic needs. They wished to additionally enforce a "natural" order on the way transactions were executed.

When we discussed transactions, particularly the isolation semantic called serializability, there were some questions in class about the order in which concurrent transactions are executed by the database. Remember that with *serializability*, the semantic is that concurrent transactions appear to have been executed in *some* sequential order, but there is no restriction on which order. For example, if I submit a transaction between accounts A and B, and you submit a transaction between accounts A and C, then either yours will go first and mine will be attempted second (and go through if there's still enough money in A after you've done yours), or vice versa. But unless one of us waits for the outcome of the transaction before the other issues his/her transaction, the order in which the transactions will be applied by the DB is undetermined. That can get a bit tricky, because if I am sitting next to you as we issue these transactions, and I clearly witness that you submitted your transaction way after I submitted mine, then I could get annoyed if my transaction is rejected due to insufficient funds when yours went through.

Ideally, we would want the database to commit transactions in an order that is consistent with the **real-time ordering** in which they were issued. This, turns out, is not achievable, but a weaker yet still very useful version of this ordering restriction is possible, and that is what Spanner provides. The semantic is called ***external consistency***, and the goal is for the database system to commit transactions in an order that is consistent with the "commit-to-begin" rule: if a transaction T2 was issued *after* another transaction T1 was committed, then T2's commit must be ordered after T1's commit.

## III. TrueTime

The first thing that one needs before one can implement external consistency, is a notion of real time. It turns out that getting a global notion of "real time" in a distributed system is difficult. The reason is that the clocks on individual machines are independent and evolve independently, over time drifting from each other. We can synchronize them periodically to a reference time keeper – and that is what all of our devices do periodically using the network time protocol (NTP) – but the errors can still be large (tens of milliseconds for NTP) and it's hard to get definite error bounds on the error between two clocks.

To address the problem, Google developed *TrueTime*, a time service that they deployed in all of their data centers to keep accurate time. TrueTime relies upon expensive hardware and carefully crafted protocols to accurately bound the synchronization error on any machine in the data center w.r.t. to a set of reference clocks, which themselves are very tightly synchronized.

TrueTime architecture:

- In each data center, they have a set of expensive, but very accurate, atomic clocks. They use these clocks as *reference clocks*.
- On each machine, they have a regular clock (e.g., quartz), which is inexpensive but inexact. The machine also runs a *time daemon*, which selects multiple reference clocks from multiple data centers and performs a particular algorithm to synchronize its own clock with theirs. During the synchronization protocol, the error is bounded very precisely (to a few milliseconds in general).
- Between synchronizations, the time daemon will accumulate error: about 20usec/second is applied, which is high enough to capture most drift that common clocks have.

TrueTime interface (used by Spanner and other TrueTime applications):

- TTInterval tt = TT.now();
- tt.latest – tt.earliest = epsilon, the instantaneous error bound
- In practice, epsilon sawtooths between 1ms and 6ms.

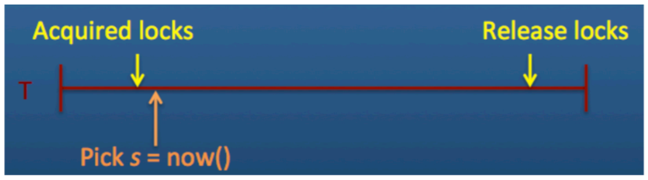
## IV Externally Consistent Transactions in Spanner

The following notes, which are adapted from Steve Gribble's Spanner design notes at the University of Washington (https://courses.cs.washington.edu/courses/csep552/13sp/lectures/6/spanner.pdf), describe how Spanner uses TrueTime to implement externally consistent transactions. In addition to these notes, please view the Spanner OSDI talk: https://www.usenix.org/node/170855.

To implement external consistency, we need to: (1) assign a **timestamp to a transaction** and (2) make sure that the timestamp is **consistent with global time**.

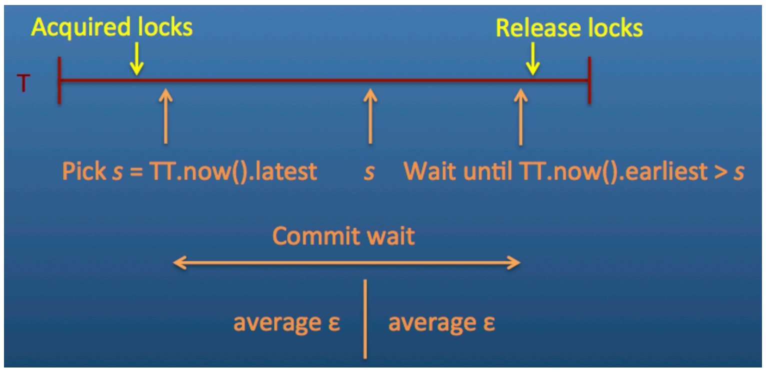
**(1) Assigning a timestamp to a transaction**

Recall how locking works in a typical transactional system: the transaction begins, then there's a period where you gather locks, then a period where locks are held, then a period where you release locks, and finally the transaction ends. We can assign a timestamp *anytime between (all locks held) and (any locks released)*, as shown in the figure to the right. Q: Why?



**(2) Ensuring consistency with global time**

Once we've picked this timestamp, we need to ensure that the transaction timestamp is consistent with global time. In other words, nobody else should be able to see the transaction side-effects until after s.latest. To do this, need to introduce a "*commit wait*" period before the locks are released.


Acquired locks
Release locks
T
Pick *s* = TT.now().latest
*s*
Wait until TT.now().earliest > *s*
Commit wait
average ε
average ε

**Commit wait** is what enforces external consistency: if the start of T2 occurs after the commit of T1, then the commit timestamp of T2 > commit timestamp of T1. Proof is in the Spanner paper.

**Picking a timestamp during 2-Phase commit**
Recall that Spanner uses 2PC to perform distributed transactions across tablets. So in picking a timestamp for a transaction, one has to take into account all participants. The 2PC coordinator gathers a number of TrueTime timestamps: (a) the prepare timestamps from non-coordinators, (b) the timestamp that the coordinator received the commit message from the client, TTcommit. The coordinator then chooses overall transaction timestamp to be greater than the prepare timestamps, greater than TTcommit.latest, and greater than any timestamps assigned to earlier transactions. Does commit wait based on this maximum.

**Implications of commit wait**
- the larger the uncertainty bound from TrueTime, the longer commit wait period you get
- commit wait will slow down dependent transactions, since locks are held during commit wait
- so, as time gets less certain, Spanner gets slower (!!). View talk or read paper for an evaluation.